REUVEN M. LERNER

Flask

Love Python, but don't want the overhead of a large Web framework? Try Flask, a lean, powerful microframework.

Let's face it, the Web has gotten big and complicated. No longer is it really possible for someone to be the "Webmaster", as we used to say back in the olden days of Web development. Today, we have front-end developers, back-end developers, system administrators, graphic designers, writers and any number of other jobs associated with the Web. Those of us fortunate enough to know a few of these things call ourselves "full-stack Web developers", but even full-stack developers need other people, with other talents, in order to get a Web application up and running.

As the Web has become more complex, so have the frameworks we use to develop applications. Once, we could put up a simple application in a matter of minutes by writing a CGI program. Later, it was enough to slap together a few pages of PHP or perhaps even a template that mixed HTML with a higher-level language. But then came the frameworks—first the big ones, from the Java and .NET worlds, and then the open-source ones, particularly Rails (for Ruby) and Django (for Python).

These frameworks are totally amazing, and they do just about everything you ever would want from a Web-development framework. But over time, these frameworks—developed in order to get away from large, do-everything frameworks from the world of enterprise software—have become big. I won't use the term "bloated", because the fact is that I believe most framework maintainers are doing a good job of balancing the core needs and functionality with optional extras.

However, there are times when you want the best of all worlds—the ease and speed of creating something without a big framework, while still enjoying the benefits that a framework can provide. This is where "micro-frameworks" can suit your needs perfectly. For example, when creating the site that powers my consulting Web site (**http://lerner.co.il**), I

wanted there to be some dynamic content and also to be able to program things. But, I wasn't about to fire up a full instance of Rails or Django just for that.

One of the first, and best-known, microframeworks is Sinatra, which I covered in this column several years ago. Sinatra is written in Ruby, which makes it a great alternative to Rails for smaller projects. But if you're a Python developer, and particularly if you want to make use of the terrific Python infrastructure and community, you actually have several options from which to choose.

Perhaps the best known and most fully featured microframework for Python is Flask, written by Armin Ronacher and other members of the international "Pocoo" team of Python developers. There are other microframeworks for Python, such as Bottle, but Flask seems to do a good job of balancing ease of use, a small core, oodles of features, a distinctly Python-like feeling when developing sites in it and a large array of extensions that make it easy to add all sorts of functionality without writing it yourself.

So in this article, I take a brief tour of Flask and show how it can make life quite easy for Web developers. I've already incorporated Flask into the curriculum of some of my Python courses, not only because it allows us to get up and running quickly, but also because I find that the design reinforces the coding style Python developers should aim to attain.

Starting with Flask

Assuming that you have pip (the modern Python installation program) on your computer, you can install Flask with:

```
pip install flask
```

(Depending on the permissions of your computer, you might need to install the above as root.)

With that package (and its dependencies) in place, you're ready to start developing. Create a new Python program that contains the following:

```
#!/usr/bin/env python

from flask import Flask
app = Flask(__name__)

@app.route("/")
def foo():
    return "Hello, world...!"

app.run(debug=True)
```

Let's go through this program (which I have called simple.py), line by line, to see what it does before

Listing 1. simple.py

```
#!/usr/bin/env python

from flask import Flask, render_template, request
app = Flask(__name__)

@app.route("/blah")
@app.route("/")
def foo():
    return render_template('foo.html')

@app.route("/submit", methods=["POST"])
def submit():
    username = request.form['name']
    return "Thank you for submitting a form, %s." % username

app.run(debug=True)
```

Listing 2. foo.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello!</title>
  </head>
  <body>
    <h1>Hello!</h1>
    <p>Test paragraph</p>

    <form method="POST" action="/submit">
      <p>Name: <input type="text" name="name" /></p>
      <p><input type="submit" value="Send your name" /></p>
    </form>

  </body>
</html>
```

you run it. The first line is actually not surprising to anyone with Python experience; it simply means that you want to open the "flask" module or package and put the "Flask" class defined there in the current namespace. With that done, you now can create an instance of Flask, which you do using the built-in `__name__`, which, of course, will be the string `__main__` on all programs run directly from the command line.

Now the real magic begins, with the line starting with `@app.route`. The @ at the start of the lines indicates that this is a decorator. This isn't the appropriate place to go into detail on what decorators do,

but suffice it to say that this allows `app.route` to execute both before the `foo() function` is defined, and also each time it is executed. Routes are just one place in Flask that use decorators in this way; you also can use them to ensure that certain actions happen before or after others.

You also can use multiple decorators on a single function definition. Thus, if you (for whatever reason) want two different URLs to invoke the same function, just stack the decorators:

```
@app.route("/foo")
@app.route("/")
def foo():
    return "Hello, world!"
```

Next, you can see that the function "foo" takes no parameters and is a normal Python function. The only unusual thing about this function is that you don't invoke it directly. Rather, the Flask framework invokes your function for you, based on the URL to which the user has navigated. So when the user goes to "/", which has been registered with `app.route`, Flask knows to invoke this function, and it does so. The string that is returned by the function is then returned to the user's browser.

Finally, you tell your application, which you created with the call to `Flask(__name__)` at the top of the file, that it should run. You could do that just by invoking `app.run()`, but by passing `debug=True` as a keyword parameter, you gain a number of things, including automatic reloading of files without restarting the server, and a browser-based debugger and console if and when things go wrong.

If you now invoke simple.py:

```
$ ./simple.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

the server is waiting on port 5000. You can point your browser to http://localhost:5000 and immediately see "Hello, world!" It's not the fanciest of responses, but not a bad one.

I'm probably jumping the gun a bit, but the Web-based console is pretty snazzy, and I think it really can help in debugging. Instead of the "return" line shown above, in which you return a string, replace it with the line:

```
return "a" + 5
```

No, this line isn't valid Python, but that's just the point. You're deliberately forcing an error. Now, go back to the "/" URL in your browser,

and you should see a long stack trace with error messages. So far, that's not too exciting. But if you move your mouse cursor to the right side of the darker lines (that is, those containing source code), you'll see two icons appear. The rightmost icon (of a text file) will display the Python source code that was executing when the error occurred.

Far more powerful is what happens if you click on the darker (terminal-like) icon. A full-fledged Python interpreter and console opens up, containing the variables and functions that were defined at that point in the stack. You can open a console for any point in the stack trace, explore the stack frame and its variables, and figure out what went wrong in your program.

Simple Templates

Returning strings of text, or even HTML, from within a function is good for a simple demo, but quickly becomes tedious. Flask's JinJa2 templates support not only HTML, but also a Python-like syntax that you can embed inside the templates. For now though, let's ignore JinJa2's capabilities and create a simple form you can submit.

First and foremost, if you are going to use JinJa2 templates, you need to create a subdirectory called "templates". This subdirectory should exist in the same directory as simple.py. Inside this templates directory, let's create a very simple HTML file:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello!</title>
  </head>
  <body>
    <h1>Hello!</h1>
    <p>Test paragraph</p>
  </body>
</html>
```

As you can see, this template won't do much. But the fact that it's a full-fledged HTML file means you suddenly can have all of the text, CSS, links and JavaScript that you want, without having to stuff it inside a triple-quoted string in your function.

As things currently stand, the template (in templates/foo.html) isn't going to be called by your "foo" function. You need to change the function such that it invokes the template. You do this with:

```
@app.route("/foo")
@app.route("/")
def foo():
    return render_template('index.html')
```

The `render_template` function is defined in the `flask` module, which means you'll need to change your `import` statement at the top of the program too:

```
from flask import Flask, render_template
```

Once you've done that, you can reload the page, and—*voilà!*—the template is rendered, as you hoped.

Handling Forms

It's pretty typical for sites to have one or more HTML forms. You've already seen just about everything you need in order for Flask to process forms, believe it or not. All you need to do now is modify your template so that it contains an HTML form, write a function that is invoked with POST, and then grab the submitted form element and do something with it. Each of these three steps is fairly easy with Flask.

First, add an HTML form to your template:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello!</title>
  </head>
  <body>
    <h1>Hello!</h1>
    <p>Test paragraph</p>

    <form method="POST" action="/submit">
      <p>Name: <input type="text" name="name" /></p>
      <p><input type="submit" value="Send your name" /></p>
    </form>

  </body>
</html>
```

This indicates that the form will be submitted using the POST method to the "/submit" URL. In addition to a "submit" button, the form consists of a single text field, called "name". If you click on the "submit" button, you get an error message:

```
The requested URL was not found on
the server. If you entered the
URL manually please check your
spelling and try again.
```

That's not a surprise, given that you haven't defined a route for it, let alone a function. Let's add that to simple.py:

```
@app.route("/submit")
def submit():
    return "Thank you for submitting a form."
```

If you go back to "/" and enter your name, then click on "submit", you get

the following error:

```
Method Not Allowed

The method is not allowed for the requested URL.
```

Notice that the problem here isn't that Flask doesn't recognize the route. Rather, the route doesn't know how to handle a POST request. That's because routes in Flask are assumed to handle GET, unless you specify otherwise. You can do that by passing the "methods" parameter to your route, specifying a list of methods (as strings) that are acceptable:

```
@app.route("/submit", methods=["POST"])
def submit():
    return "Thank you for submitting a form."
```

Sure enough, if you submit the form, you get the text back. But this text is rather generic. It would be nice to acknowledge the user's name, given that he or she went through the trouble of providing it. You can grab the user's name, as well as any other form parameters, via the "request"

object that Flask makes available to you. `request.form` is a dictionary-like object that lets you query the form via key names (as strings). You need to import "request" from the "flask" package:

```
from flask import Flask, render_template, request
```

And, then you can do this:

```
@app.route("/submit", methods=["POST"])

def submit():

    username = request.form['name']

    return "Thank you for submitting a form, %s." % username
```

If you're thinking this all seems very simple—well, that's precisely the point. Flask is there to let you run ahead quickly using the Python you already know to create simple but interesting Web applications.

Conclusion

If you are familiar with Python, want to create Web applications and don't want the overhead of a large framework like Django, you might well want to consider Flask. The core framework is (as you saw here) easy to get up and running, and the extensions make it extremely flexible and powerful.■

Reuven M. Lerner, a longtime Web developer, consultant and trainer, is completing his PhD in learning sciences at Northwestern University. You can learn about his on-line programming courses, subscribe to his newsletter or contact him at http://lerner.co.il.

Send comments or feedback via http://www.linuxjournal.com/contact or to ljeditor@linuxjournal.com.

Resources

The Flask home page is at **http://flask.pocoo.org**. This includes links to the code, documentation, tutorials, examples and an official list of Flask extensions. The documentation is excellent, and it should provide anyone with even a bit of Python knowledge with good understanding of Flask.

Miguel Grinberg, who recently authored a book about Flask for O'Reilly (which I haven't yet had a chance to read or review), has written an excellent Flask "mega-tutorial" that complements the official one very nicely: **http://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world**.

A video of Miguel's tutorial at PyCon 2014 is available at **https://www.youtube.com/watch?v=FGrlyBDQLPg**.